fminunc

Find minimum of unconstrained multivariable function

Nonlinear programming solver.

Finds the minimum of a problem specified by

$$\min_x f(x)$$

where *f*(*x*) is a function that returns a scalar.

x is a vector or a matrix; see Matrix Arguments.

Syntax

```
x = fminunc(fun,x0)
x = fminunc(fun,x0,options)
x = fminunc(problem)
[x,fval] = fminunc( __ )
[x,fval,exitflag,output] = fminunc( __ )
[x,fval,exitflag,output,grad,hessian] = fminunc( __ )
```

example example example example example

Description

`x = fminunc(fun,x0)` starts at the point `x0` and attempts to find a local minimum `x` of the function described in `fun`. The point `x0` can be a scalar, vector, or matrix. example

> **Note:** Passing Extra Parameters explains how to pass extra parameters to the objective function and nonlinear constraint functions, if necessary.
>
> `fminunc` is for nonlinear problems without constraints. If your problem has constraints, generally use `fmincon`. See Optimization Decision Table.

`x = fminunc(fun,x0,options)` minimizes `fun` with the optimization options specified in `options`. Use `optimoptions` to set these options. example

`x = fminunc(problem)` finds the minimum for `problem`, where `problem` is a structure described in Input Arguments. Create the `problem` structure by exporting a problem from Optimization app, as described in Exporting Your Work. example

`[x,fval] = fminunc( __ )`, for any syntax, returns the value of the objective function `fun` at the solution `x`. example

`[x,fval,exitflag,output] = fminunc( __ )` additionally returns a value `exitflag` that describes the exit condition of `fminunc`, and a structure `output` with information about the optimization process. example

`[x,fval,exitflag,output,grad,hessian] = fminunc( __ )` additionally returns:

- `grad` — Gradient of `fun` at the solution `x`.
- `hessian` — Hessian of `fun` at the solution `x`. See fminunc Hessian.

Examples

collapse all

Minimize a Polynomial

Minimize the function $f(x) = 3x_1^2 + 2x_1x_2 + x_2^2 - 4x_1 + 5x_2$.

Write an anonymous function that calculates the objective.

```
fun = @(x)3*x(1)^2 + 2*x(1)*x(2) + x(2)^2 - 4*x(1) + 5*x(2);
```

Call `fminunc` to find a minimum of `fun` near `[1,1]`.

```
x0 = [1,1];
[x,fval] = fminunc(fun,x0);
```

After a few iterations, `fminunc` returns the solution, `x`, and the value of the function at `x`, `fval`.

```
x,fval

x =

    2.2500   -4.7500


fval =

  -16.3750
```

Supply the Gradient

`fminunc` can be faster and more reliable when you provide derivatives.

Write an objective function that returns the gradient as well as the function value. Use the conditionalized form described in Including Gradients and Hessians. The objective function is Rosenbrock's function,

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + (1 - x_1)^2,$$

which has gradient

$$\nabla f(x) = \begin{bmatrix} -400\left(x_2 - x_1^2\right)x_1 - 2\left(1 - x_1\right) \\ 200\left(x_2 - x_1^2\right) \end{bmatrix}.$$

```
function [f,g] = rosenbrockwithgrad(x)
% Calculate objective f
f = 100*(x(2) - x(1)^2)^2 + (1-x(1))^2;

if nargout > 1 % gradient required
    g = [-400*(x(2)-x(1)^2)*x(1)-2*(1-x(1));
        200*(x(2)-x(1)^2)];
end
```

Save this code as a file on your MATLAB® path, named `rosenbrockwithgrad.m`.

Create options to use the objective function's gradient. Also, set the algorithm to `'trust-region'`.

```
options = optimoptions('fminunc','Algorithm','trust-region','SpecifyObjectiveGradient',true);
```

Set the initial point to `[-1,2]`. Then call `fminunc`.

```
x0 = [-1,2];
fun = @rosenbrockwithgrad;
x = fminunc(fun,x0,options)
```

```
Local minimum found.

Optimization completed because the size of the gradient is less than
the default value of the function tolerance.

<stopping criteria details>


x =

    1.0000    1.0000
```

Use a Problem Structure

Solve the same problem as in Supply the Gradient using a problem structure instead of separate arguments.

Write an objective function that returns the gradient as well as the function value. Use the conditionalized form described in Including Gradients and Hessians. The objective function is Rosenbrock's function,

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + (1 - x_1)^2,$$

which has gradient

$$\nabla f(x) = \begin{bmatrix} -400\left(x_2 - x_1^2\right)x_1 - 2\left(1 - x_1\right) \\ 200\left(x_2 - x_1^2\right) \end{bmatrix}.$$

```
function [f,g] = rosenbrockwithgrad(x)
% Calculate objective f
f = 100*(x(2) - x(1)^2)^2 + (1-x(1))^2;

if nargout > 1 % gradient required
    g = [-400*(x(2)-x(1)^2)*x(1)-2*(1-x(1));
        200*(x(2)-x(1)^2)];
end
```

Save this code as a file on your MATLAB path, named `rosenbrockwithgrad.m`.

Create options to use the objective function's gradient. Also, set the algorithm to `'trust-region'`.

```
options = optimoptions('fminunc','Algorithm','trust-region','SpecifyObjectiveGradient',true);
```

Create a problem structure including the initial point `x0 = [-1,2]`.

```
problem.options = options;
problem.x0 = [-1,2];
problem.objective = @rosenbrockwithgrad;
problem.solver = 'fminunc';
```

Solve the problem.

```
x = fminunc(problem)
```

```
Local minimum found.

Optimization completed because the size of the gradient is less than
the default value of the function tolerance.

<stopping criteria details>


x =

    1.0000    1.0000
```

Obtain the Optimal Objective Function Value

Find both the location of the minimum of a nonlinear function and the value of the function at that minimum.

The objective function is

$$f(x) = x(1)e^{-\|x\|_2^2} + \|x\|_2^2/20.$$

```
fun = @(x)x(1)*exp(-(x(1)^2 + x(2)^2)) + (x(1)^2 + x(2)^2)/20;
```

Find the location and objective function value of the minimizer starting at `x0 = [1,2]`.

```
x0 = [1,2];
[x,fval] = fminunc(fun,x0)
```

```
Local minimum found.

Optimization completed because the size of the gradient is less than
the default value of the function tolerance.

<stopping criteria details>


x =

   -0.6691    0.0000


fval =

   -0.4052
```

Examine the Solution Process

Choose `fminunc` options and outputs to examine the solution process.

Open This Example

Set options to obtain iterative display and use the `'quasi-newton'` algorithm.

```
options = optimoptions(@fminunc,'Display','iter','Algorithm','quasi-newton');
```

The objective function is

$$f(x) = x(1)e^{-\|x\|_2^2} + \|x\|_2^2/20.$$

```
fun = @(x)x(1)*exp(-(x(1)^2 + x(2)^2)) + (x(1)^2 + x(2)^2)/20;
```

Start the minimization at `x0 = [1,2]`, and obtain outputs that enable you to examine the solution quality and process.

```
x0 = [1,2];
[x,fval,exitflag,output] = fminunc(fun,x0,options)
```

```
                                                        First-order
 Iteration  Func-count       f(x)        Step-size       optimality
     0           3         0.256738                          0.173
     1           6         0.222149             1            0.131
     2           9          0.15717             1            0.158
     3          18        -0.227902      0.438133            0.386
     4          21        -0.299271             1             0.46
     5          30        -0.404028      0.102071           0.0458
     6          33        -0.404868             1           0.0296
     7          36        -0.405236             1          0.00119
     8          39        -0.405237             1         0.000252
     9          42        -0.405237             1         7.97e-07

Local minimum found.

Optimization completed because the size of the gradient is less than
the default value of the optimality tolerance.




x =

   -0.6691    0.0000


fval =

   -0.4052


exitflag =

     1


output =

       iterations: 9
        funcCount: 42
         stepsize: 2.9343e-04
     lssteplength: 1
    firstorderopt: 7.9721e-07
        algorithm: 'quasi-newton'
          message: 'Local minimum found....'
```

- The exit flag 1 shows that the solution is a local optimum.
- The `output` structure shows the number of iterations, number of function evaluations, and other information.
- The iterative display also shows the number of iterations and function evaluations.

Related Examples

- Unconstrained Optimization

Input Arguments

collapse all

`fun` — Function to minimize
function handle | function name

Function to minimize, specified as a function handle or function name. `fun` is a function that accepts a vector or array `x` and returns a real scalar `f`, the objective function evaluated at `x`.

Specify `fun` as a function handle for a file:

```
x = fmincon(@myfun,x0,A,b)
```

where `myfun` is a MATLAB function such as

```
function f = myfun(x)
f = ...            % Compute function value at x
```

You can also specify `fun` as a function handle for an anonymous function:

```
x = fmincon(@(x)norm(x)^2,x0,A,b);
```

If you can compute the gradient of `fun` *and* the `SpecifyObjectiveGradient` option is set to `true`, as set by

```
options = optimoptions('fminunc','SpecifyObjectiveGradient',true)
```

then `fun` must return the gradient vector `g(x)` in the second output argument.

If you can also compute the Hessian matrix *and* the `HessianFcn` option is set to `'objective'` via `options = optimoptions('fminunc','HessianFcn','objective')` *and* the `Algorithm` option is set to `'trust-region-reflective'`, `fun` must return the Hessian value `H(x)`, a symmetric matrix, in a third output argument. `fun` can give a sparse Hessian. See Hessian for fminunc trust-region or fmincon trust-region-reflective algorithms for details.

The `trust-region-reflective` algorithm allows you to supply a Hessian multiply function. This function gives the result of a Hessian-times-vector product without computing the Hessian directly. This can save memory. See Hessian Multiply Function.

Example: `fun = @(x)sin(x(1))*cos(x(2))`

Data Types: `char` | `function_handle`

`x0` — Initial point

real vector | real array

Initial point, specified as a real vector or real array. Solvers use the number of elements in, and size of, `x0` to determine the number and size of variables that `fun` accepts.

Example: `x0 = [1,2,3,4]`

Data Types: `double`

`options` — Optimization options

output of `optimoptions` | structure such as `optimset` returns

Optimization options, specified as the output of `optimoptions` or a structure such as `optimset` returns.

Some options apply to all algorithms, and others are relevant for particular algorithms. See Optimization Options Reference for detailed information.

Some options are absent from the `optimoptions` display. These options are listed in italics. For details, see View Options.

All Algorithms

`Algorithm` If you use `optimset` (not recommended, see Choose Between optimoptions and optimset), use `LargeScale` instead of `Algorithm`.	Choose the `fminunc` algorithm. Choices are `'trust-region'` (default) or `'quasi-newton'`. The `'trust-region'` algorithm requires you to provide the gradient (see the description of `fun`), or else `fminunc` uses the `'quasi-newton'` algorithm. For information on choosing the algorithm, see Choosing the Algorithm.
`CheckGradients`	Compare user-supplied derivatives (gradient of objective) to finite-differencing derivatives. Choices are `false` (default) or `true`.
Diagnostics	Display diagnostic information about the function to be minimized or solved. Choices are `'off'` (default) or `'on'`.
DiffMaxChange	Maximum change in variables for finite-difference gradients (a positive scalar). The default is `Inf`.
DiffMinChange	Minimum change in variables for finite-difference gradients (a positive scalar). The default is `0`.

`Display`	Level of display (see Iterative Display): • `'off'` or `'none'` displays no output. • `'iter'` displays output at each iteration, and gives the default exit message. • `'iter-detailed'` displays output at each iteration, and gives the technical exit message. • `'notify'` displays output only if the function does not converge, and gives the default exit message. • `'notify-detailed'` displays output only if the function does not converge, and gives the technical exit message. • `'final'` (default) displays only the final output, and gives the default exit message. • `'final-detailed'` displays only the final output, and gives the technical exit message.

`FiniteDifferenceStepSize`	Scalar or vector step size factor for finite differences. When you set `FiniteDifferenceStepSize` to a vector `v`, forward finite differences steps `delta` are
	`delta = v.*sign′(x).*max(abs(x),TypicalX);`
	where `sign′(x) = sign(x)` except `sign′(0) = 1`. Central finite differences are
	`delta = v.*max(abs(x),TypicalX);`
	Scalar `FiniteDifferenceStepSize` expands to a vector. The default is `sqrt(eps)` for forward finite differences, and `eps^(1/3)` for central finite differences.
	The trust-region algorithm uses `FiniteDifferenceStepSize` only when `CheckGradients` is set to `true`.

`FiniteDifferenceType`	Finite differences, used to estimate gradients, are either `'forward'` (the default), or `'central'` (centered). `'central'` takes twice as many function evaluations, but should be more accurate. The trust-region algorithm uses `FiniteDifferenceType` only when `CheckGradients` is set to `true`.
`FunctionTolerance`	Termination tolerance on the function value, a positive scalar. The default is `1e-6`. See Tolerances and Stopping Criteria.
FunValCheck	Check whether objective function values are valid. The default setting, `'off'`, does not perform a check. The `'on'` setting displays an error when the objective function returns a value that is `complex`, `Inf`, or `NaN`.
`LargeScale` If you use `optimoptions` (recommended, see Choose Between optimoptions and optimset), use `Algorithm` instead of `LargeScale`.	Choose the algorithm. When set to the default `'on'`, use the `'trust-region'` algorithm if possible. When set to `'off'`, use the `'quasi-newton'` algorithm. The `'trust-region'` algorithm requires you to provide the gradient (see the description of `fun`). Otherwise, `fminunc` uses the `'quasi-newton'` algorithm. For more information, see Choosing the Algorithm.
`MaxFunctionEvaluations`	Maximum number of function evaluations allowed, a positive integer. The default value is `100*numberOfVariables`. See Tolerances and Stopping Criteria and Iterations and Function Counts.
`MaxIterations`	Maximum number of iterations allowed, a positive integer. The default value is `400`. See Tolerances and Stopping Criteria and Iterations and Function Counts.
`OptimalityTolerance`	Termination tolerance on the first-order optimality, a positive scalar. The default is `1e-6`. See First-Order Optimality Measure.
`OutputFcn`	Specify one or more user-defined functions that an optimization function calls at each iteration, either as a function handle or as a cell array of function handles. The default is none (`[]`). See Output Function.
`PlotFcn`	Plot various measures of progress while the algorithm executes. Select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[]`). • `@optimplotx` plots the current point. • `@optimplotfunccount` plots the function count. • `@optimplotfval` plots the function value. • `@optimplotstepsize` plots the step size. • `@optimplotfirstorderopt` plots the first-order optimality measure. For information on writing a custom plot function, see Plot Functions.

`SpecifyObjectiveGradient`

Gradient for the objective function defined by the user. See the description of `fun` to see how to define the gradient in `fun`. Set to `true` to have `fminunc` use a user-defined gradient of the objective function. The default `false` causes `fminunc` to estimate gradients using finite differences. You must provide the gradient, and set `SpecifyObjectiveGradient` to `true`, to use the trust-region algorithm. This option is not required for the quasi-Newton algorithm.

`StepTolerance`

Termination tolerance on `x`, a positive scalar. The default value is `1e-6`. See Tolerances and Stopping Criteria.

`TypicalX`

Typical `x` values. The number of elements in `TypicalX` is equal to the number of elements in `x0`, the starting point. The default value is `ones(numberofvariables,1)`. `fminunc` uses `TypicalX` for scaling finite differences for gradient estimation.

The `trust-region` algorithm uses `TypicalX` only for the `CheckGradients` option.

`trust-region` Algorithm

`HessianFcn`

If set to `[]` (default), `fminunc` approximates the Hessian using finite differences.

If set to `'objective'`, `fminunc` uses a user-defined Hessian for the objective function. The Hessian is the third output of the objective function (see `fun`).

`HessianMultiplyFcn`

Function handle for Hessian multiply function. For large-scale structured problems, this function computes the Hessian matrix product H∗Y without actually forming H. The function is of the form

```
W = hmfun(Hinfo,Y)
```

where `Hinfo` contains the matrix used to compute H∗Y.

The first argument is the same as the third argument returned by the objective function `fun`, for example

```
[f,g,Hinfo] = fun(x)
```

Y is a matrix that has the same number of rows as there are dimensions in the problem. The matrix `W = H∗Y`, although H is not formed explicitly. `fminunc` uses `Hinfo` to compute the preconditioner. For information on how to supply values for any additional parameters `hmfun` needs, see Passing Extra Parameters.

> **Note** To use the `HessianMultiplyFcn` option, `HessianFcn` must be set to `[]`.

For an example, see Minimization with Dense Structured Hessian, Linear Equalities.

`HessPattern`

Sparsity pattern of the Hessian for finite differencing. Set `HessPattern(i,j) = 1` when you can have $\partial^2 \mathtt{fun}/\partial \mathtt{x(i)}\partial \mathtt{x(j)} \neq 0$. Otherwise, set `HessPattern(i,j) = 0`.

Use `HessPattern` when it is inconvenient to compute the Hessian matrix H in `fun`, but you can determine (say, by inspection) when the `i`th component of the gradient of `fun` depends on `x(j)`. `fminunc` can approximate H via sparse finite differences (of the gradient) if you provide the *sparsity structure* of H as the value for `HessPattern`. In other words, provide the locations of the nonzeros.

When the structure is unknown, do not set `HessPattern`. The default behavior is as if `HessPattern` is a dense matrix of ones. Then `fminunc` computes a full finite-difference approximation in each iteration. This computation can be expensive for large problems, so it is usually better to determine the sparsity structure.

`MaxPCGIter`

Maximum number of preconditioned conjugate gradient (PCG) iterations, a positive scalar. The default is `max(1,floor(numberOfVariables/2))`. For more information, see Trust Region Algorithm.

`PrecondBandWidth`

Upper bandwidth of preconditioner for PCG, a nonnegative integer. By default, `fminunc` uses diagonal preconditioning (upper bandwidth of 0). For some problems, increasing the bandwidth reduces the number of PCG iterations. Setting `PrecondBandWidth` to `Inf` uses a direct factorization (Cholesky) rather than the conjugate gradients (CG). The direct factorization is computationally more expensive than CG, but produces a better quality step towards the solution.

`SubproblemAlgorithm`

Determines how the iteration step is calculated. The default, `'cg'`, takes a faster but less accurate step than `'factorization'`. See fminunc trust-region Algorithm.

`TolPCG`

Termination tolerance on the PCG iteration, a positive scalar. The default is `0.1`.

`quasi-newton` Algorithm

HessUpdate

Method for choosing the search direction in the Quasi-Newton algorithm. The choices are:

- `'bfgs'`, the default
- `'dfp'`
- `'steepdesc'`

See Quasi-Newton Algorithm and Hessian Update for a description of these methods.

InitialHessMatrix
This option will be removed in a future release.

Initial quasi-Newton matrix. This option is available only if you set `InitialHessType` to `'user-supplied'`. In that case, you can set `InitialHessMatrix` to one of the following:

- A positive scalar — The initial matrix is the scalar times the identity.
- A vector of positive values — The initial matrix is a diagonal matrix with the entries of the vector on the diagonal. This vector should be the same size as the `x0` vector, the initial point.

InitialHessType
This option will be removed in a future release.

Initial quasi-Newton matrix type. The options are:

- `'identity'`
- `'scaled-identity'`, the default
- `'user-supplied'` — See `InitialHessMatrix`

`ObjectiveLimit`

A tolerance (stopping criterion) that is a scalar. If the objective function value at an iteration is less than or equal to `ObjectiveLimit`, the iterations halt because the problem is presumably unbounded. The default value is `-1e20`.

`UseParallel`

When `true`, `fminunc` estimates gradients in parallel. Disable by setting to the default, `false`. `trust-region-reflective` requires a gradient in the objective, so `UseParallel` does not apply. See Parallel Computing.

Example: `options = optimoptions('fminunc','SpecifyObjectiveGradient',true)`

`problem` — Problem structure
structure

Problem structure, specified as a structure with the following fields:

Field Name	Entry
`objective`	Objective function
`x0`	Initial point for x
`solver`	`'fminunc'`
`options`	Options created with `optimoptions`

The simplest way to obtain a `problem` structure is to export the problem from the Optimization app.

Data Types: `struct`

Output Arguments

collapse all

`x` — Solution
real vector | real array

Solution, returned as a real vector or real array. The size of x is the same as the size of `x0`. Typically, x is a local solution to the problem when `exitflag` is positive. For information on the quality of the solution, see When the Solver Succeeds.

`fval` — Objective function value at solution
real number

Objective function value at the solution, returned as a real number. Generally, `fval` = `fun(x)`.

`exitflag` — Reason **`fminunc`** stopped
integer

Reason `fminunc` stopped, returned as an integer.

1	Magnitude of gradient is smaller than the `OptimalityTolerance` tolerance.
2	Change in `x` was smaller than the `StepTolerance` tolerance.
3	Change in the objective function value was less than the `FunctionTolerance` tolerance.
5	Predicted decrease in the objective function was less than the `FunctionTolerance` tolerance.
0	Number of iterations exceeded `MaxIterations` or number of function evaluations exceeded `MaxFunctionEvaluations`.
-1	Algorithm was terminated by the output function.
-3	Objective function at current iteration went below `ObjectiveLimit`.

`output` — Information about the optimization process
structure

Information about the optimization process, returned as a structure with fields:

`iterations`	Number of iterations taken
`funcCount`	Number of function evaluations
`firstorderopt`	Measure of first-order optimality
`algorithm`	Optimization algorithm used
`cgiterations`	Total number of PCG iterations (`'trust-region'` algorithm only)
`lssteplength`	Size of line search step relative to search direction (`'quasi-newton'` algorithm only)
`stepsize`	Final displacement in `x`
`message`	Exit message

`grad` — Gradient at the solution
real vector

Gradient at the solution, returned as a real vector. `grad` gives the gradient of `fun` at the point `x(:)`.

`hessian` — Approximate Hessian
real matrix

Approximate Hessian, returned as a real matrix. For the meaning of `hessian`, see Hessian.

More About

collapse all

Algorithms

Trust Region Algorithm

The `trust-region` algorithm requires that you supply the gradient in `fun` and set `SpecifyObjectiveGradient` to `true` using `optimoptions`. This algorithm is a subspace trust-region method and is based on the interior-reflective Newton method described in [2] and [3]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See `fminunc trust-region` Algorithm, Trust-Region Methods for Nonlinear Minimization and Preconditioned Conjugate Gradient Method.

Quasi-Newton Algorithm

The `quasi-newton` algorithm uses the BFGS Quasi-Newton method with a cubic line search procedure. This quasi-Newton method uses the BFGS ([1],[5],[8], and [9]) formula for updating the approximation of the Hessian matrix. You can select the DFP ([4],[6], and [7]) formula, which approximates the inverse Hessian matrix, by setting the `HessUpdate` option to `'dfp'` (and the `Algorithm` option to `'quasi-newton'`). You can select a steepest descent method by setting `HessUpdate` to `'steepdesc'` (and `Algorithm` to `'quasi-newton'`), although this setting is usually inefficient. See `fminunc quasi-newton` Algorithm.

- Optimization Problem Setup
- Unconstrained Nonlinear Optimization Algorithms

References

[1] Broyden, C. G. "The Convergence of a Class of Double-Rank Minimization Algorithms." *Journal Inst. Math. Applic.*, Vol. 6, 1970, pp. 76–90.

[2] Coleman, T. F. and Y. Li. "An Interior, Trust Region Approach for Nonlinear Minimization Subject to Bounds." *SIAM Journal on Optimization*, Vol. 6, 1996, pp. 418–445.

[3] Coleman, T. F. and Y. Li. "On the Convergence of Reflective Newton Methods for Large-Scale Nonlinear Minimization Subject to Bounds." *Mathematical Programming*, Vol. 67, Number 2, 1994, pp. 189–224.

[4] Davidon, W. C. "Variable Metric Method for Minimization." *A.E.C. Research and Development Report*, ANL-5990, 1959.

[5] Fletcher, R. "A New Approach to Variable Metric Algorithms." *Computer Journal*, Vol. 13, 1970, pp. 317–322.

[6] Fletcher, R. "Practical Methods of Optimization." Vol. 1, *Unconstrained Optimization*, John Wiley and Sons, 1980.

[7] Fletcher, R. and M. J. D. Powell. "A Rapidly Convergent Descent Method for Minimization." *Computer Journal*, Vol. 6, 1963, pp. 163–168.

[8] Goldfarb, D. "A Family of Variable Metric Updates Derived by Variational Means." *Mathematics of Computing*, Vol. 24, 1970, pp. 23–26.

[9] Shanno, D. F. "Conditioning of Quasi-Newton Methods for Function Minimization." *Mathematics of Computing*, Vol. 24, 1970, pp. 647–656.

See Also

`fmincon` | `fminsearch` | `optimoptions`

Introduced before R2006a